AIviz Documentation

Each group has already been assigned a problem statement. The said algorithm has to be implemented and integrated with this platform. BFS and DFS has been integrated with the platform as an implementation demo.

A high level implementation structure of the platform is given below:
(also, the constructs that needs to be modified by you)

- **alviz.algorithm -** consists of already integrated algorithms (see BFS/DFS) and **to-be-integrated algorithms have to be placed in this package**
- **alviz.base.algorithm -** contains the implementation of base algorithm *Algorithm* which maintains four execution states: initialized, running, ended, terminated and, the getters and setters for the same. Your algorithm should inherit the abstract class *Algorithm* and implement its *execute()* method. (see BFS/DFS)

  Note that the invocation of *show()* whenever the node/edge type: *open, closed etc* is updated is important to show the step wise animation.

  Algorithm factory creates *algotype* and its corresponding *graph* for your algorithm, so do not forget to place your algorithm name in *algotype* switch construct here.

  ```
  case algorithm_name:
  al = new algorithm_name((Graph)graph)
  @Override
  public void paint()  graphCanvas.repaint();
  ;
  break;
  ```

  The description of *algotype* has been given in *AlgorithmType* class.
  Include your algorithm name and its graphclass here.

  ```
  algoname("algoname", GraphClass.GRAPH)
  ```

- **alviz.base.graph-** *Basegraph* class has all the methods that your algorithm might need to communicate with the problem graph generated by the platform. You will also find the various color coding options, to represent the runtime state of nodes and edges, here. You can define more as per your requirements.

GraphPainter takes care of rendering graph according to the assigned runtime state of nodes and edges.

You can define other runtime states of nodes and edges (in *BaseGraph* class) depending on the requirements of your algorithm.
For example, for TSP you might not want to show the edges that are not part of candidate solution but still need them for computation, you can define a runtime state for edge, say *Invisible*. While rendering the graph (see *paintEdge()* in *GraphPainter* class) do not render edges which are assigned runtime state *Invisible*.

- **alviz.graph-** *Graph* class inherits *Basegraph* and provides algorithm specific methods. You can customize *Graph* class in *alviz.graph* as per your requirements.
- **alviz.graph.factory** – contains graph construction code –
- **alviz.main** Contains GUI and application logic, and maintains application state. For TSP (all version) and SA interface, you might need to tweak the code here.
- alviz.util

Color Coding

- **OPEN**
- **CLOSED**
- **DELETED**
- **OLD**
- **RELAY**
- **BOUNDARY**
- **Path found**

etc... (Suggestions are welcome)